Exascale Programming Approaches for the Accelerated Model for Climate and Energy

Matthew R. Norman; Azamat Mametjanov; Mark Taylor

January 19, 2017

1 Overview and Scientific Impact of ACME

The Accelerated Model for Climate and Energy (ACME) is a leading-edge climate and Earth system model designed to address U.S. Department of Energy (DOE) mission needs. The ACME project involves collaboration between seven National Laboratories, the National Center for Atmospheric Research, four academic institutions, and one private-sector company. The ACME project funds ongoing efforts to ensure the model continues to make efficient use of DOE Leadership Computing Facilities (LCFs) in order to best address the three driving grand challenge science questions:

1. Water Cycle: How will more realistic portrayals of features important to the water cycle (resolution, clouds, aerosols, snowpack, river routing, land use) affect river flow and associated freshwater supplies at the watershed scale?

2. Cryosphere Systems: Could a dynamical instability in the Antarctic Ice Sheet be triggered within the next 40 years?

3. Biogeochemistry: What are the contributions and feedbacks from natural and managed systems to current greenhouse gas fluxes, and how will those factors and associated fluxes evolve in the future?

For the water cycle, the goal is to simulate the changes in the hydrological cycle with a specific focus on precipitation and surface water in orographically complex regions such as the western United States and the headwaters of the Amazon. For the cyrosphere, the objective is to examine the near-term risk of initiating the dynamic instability and onset of the collapse of the Antarctic Ice Sheet due to rapid melting by warming waters adjacent to the ice sheet grounding lines. For the biogeochemistry, the goal is to examine how coupled terrestrial and coastal ecosystems drive natural sources and sinks of carbon dioxide and methane in a warmer environment.

The ACME model is composed currently of five main components: atmosphere, ocean, land surface, sea ice, and land ice. All of these components are connected through a model coupler. The atmospheric component, the Community Atmosphere Model - Spectral Element (CAM-SE) [11, 10] is based on a local, time-explicit Spectral Element method on cubed-sphere topology with static adaptation capability. CAM-SE currently uses the hydrostatic assumption, and it includes an evolving set of physics packages. The atmospheric component is the most expensive currently. The next most expensive component is the Ocean model, the Model for Prediction Across Scales - Ocean (MPAS-O) [7], which is based on fully unstructured lower-ordered Finite-Volume approximations with split-explicit handling of the barotropic equations.

2 GPU Refactoring of ACME Atmosphere

CAM-SE can be decomposed ideologically and practically into three separate parts: dynamics, tracers, and physics. The "dynamics" are gas-only stratified fluid dynamics on a rotating sphere with a hydrostatic approximation to eliminate vertically propagating acoustic waves. The "tracers" component transports quantities used by the dynamics and physics along wind trajectories, and examples of these quantities are the three forms of water, CO2, methane, aerosols, etc. Finally, the "physics" are everything either below

spatiotemporal truncation in the dynamics or not included in the equation set to begin with. This includes eddy viscosity, gravity wave drag, shallow and deep moist convection, radiation, microphysics, and other physical phenomena. The physics and transport are operator split in an advection-reaction manner.

2.1 Mathematical Considerations and Their Computational Impacts

2.1.1 Mathematical Formulation

CAM-SE, as the name suggests, is based upon the Spectral Element (SE) numerical discretization of the underlying PDEs for stratified, hydrostatic fluid dynamics on the rotating sphere. The Spectral Element belongs the class of Finite-Element methods, or more specifically, Continuous Galerkin methods. It, therefore, uses a variational form of the PDEs, and the underlying basis in this case is 2-D tensored Lagrange interpolating polynomials on a grid of Gauss-Legendre-Lobatto (GLL) points. When integration by parts is performed, the subsequent Derichlet element boundary terms drop out due to the fact that C^0 continuity is maintained at element boundaries by a straightforward linear averaging operation.

Body integrals are solved using GLL quadrature, which in coordination with the GLL Lagrange interpolating basis, leads to a discrete orthogonality that renders the mass matrix diagonal, though it is inexactly integrated. This discrete orthogonality also simplifies the body integral calculations such that they are simply 1-D sweeps of quadrature over each of the basis functions, and this is the main calculation of the SE method. What this ends up looking like computationally is 1-D sweeps of $N \times N$ matrix-vector mulitplies (where N is the spatial order of accuracy), and indeed, the SE method can be cast in this way. Thus, the computational complexity of the SE spatial operator is DN^{D+1}, where D is the spatial dimensionality ($D = 2$ in this case), and this is generally the optimal computational complexity for a time-explicit, hyperbolic spatial operator.

The fact that the SE method at its core is essentially made up of 1-D sweeps of matrix-vector multiplies is advantageous for GPUs because this means that data fetched from DRAM will be re-used. The data access scales as N^2, while the computation scales as DN^{D+1}, making this a compute intensive operation – all the more so as N increases. On the ground level, matrix-vector multiplies always have an innermost loop that is highly dependent, and it accesses different indices of the same arrays multiple times. This means that for efficient execution on a GPU, these kernels absolutely must use CUDA Shared Memory, if only due to the fact that arrays are accessed non-contiguously in the innermost loop. Another advantage to SE methods is that they can cluster the majority of the computation into a single unbroken loop / kernel. It's good to have numerical methods that to do not inherently require a significant amount of global dependence between global data structures.

2.1.2 Grid

CAM-SE realizes spherical geometry by use of the so-called cubed-sphere [8] grid, which in this case utilizes a non-orthogonal equal-angle gnomonic projection from a cube onto the sphere. This grid is advantageous in several ways: (1) it provides nearly uniform grid spacing; (2) it avoids the strong polar singularities experienced by a latitude-longitude grid and instead has eight weaker singularities at cube corners; (3) It is logically structured, which can simplify operations like element edge averaging; and (4) it avoids conservation and / or CFL difficulties experienced by overset grids such as Yin-Yang [3]. The main disadvantageous aspect is the use of non-orthogonal coordinates, which complicates the numerics somewhat. Each of the cubed-sphere's six panels is subdivided into ne×ne elements.

2.1.3 Element Boundary Averaging

Again, the element boundaries are kept continuous by a linear averaging of the element boundary basis function coefficients (the unknowns being evolved by SE). This is the only means of element-to-element communication in the SE method, as all spatial operators must have some mechanism of communication between DOFs on the grid. Though it is low in computation, it actually consumes more time than the body integral calculations because it is heavy in terms of data movement. This averaging can easily induce a race condition in that one element's data must not yet be replaced with the linear average before the other element accesses it. To keep race conditions from occurring, the developers of CAM-SE chose to pack all element edge data into a process-wide buffer, and then during the unpacking procedure, the averaging is

performed. The benefit of this is that it makes arbitrarily unstructured meshes easy to implement. The downsides are that: (1) DRAM accesses are doubled by packing to and from yet another buffer, (2) many DRAM accesses are neither contiguous nor ordered, and (3) this often leads to having to access data owned by another core on the CPU. In hindsight, it would be better to traverse the edges themselves and average the data in place, and there are efforts to investigate this.

In parallel implementations, SE is particularly advantageous because the only communication between adjacent nodes is the element edge data that overlays a domain decomposition boundary. This is an optimal communication requirement.

2.1.4 Limiting

There are two mechanisms for limiting oscillations that inevitably resort from applying the SE method in the presence of non-smooth data. The first option is a fourth-order "hyperviscosity," which is expressed in PDE form as: $\partial q/\partial t = -\nu\nabla^4 q$, where ν is the coefficient of viscosity. This fourth-order derivative term is computed by applying a Laplacian operator twice, each application followed by an averaging of element boundary data. The hyperviscosity operator is included along with the standard advective update in the tracer transport, and therefore, the last stage of element edge data averaging of the hyperviscosity, being coupled with the advective update, is hidden. Also, it is only applied once for tracers. For the dynamics, the hyperviscosity operator is used essentially as an energy cascade closure scheme, and it must be applied multiple times (i.e., "sub-cycled") per time step in order to achieve a proper energy cascade. For the dynamics, hyperviscosity takes up more time than the fluid update step.

The other mechanism for limiting oscillations is a monotone limiter that is only applied to tracer transport. The limiter begins by gathering the maximum and minimum basis coefficient values among all neighboring elements and the element in question. Then, after the advective step takes place, a limiter routine is called that begins by replacing any data that exceeds the maximum (minimum) bounds with the maximum (minimum) value, and then proceeds to redistribute the changed mass in an optimal manner that best maintains similarity to the original data shape, and this limiter is applied in multiple iterations. There must also be included a hyperviscosity tendency along with the advective step to avoid artifacts known as "terracing" that also are common in Flux Corrective Transport methods when a viscous process is excluded [12].

2.1.5 Time Discretization

The SE method is cast into semi-discrete form, meaning the temporal derivative is left in place as the last remaining continuously differentiated term in the PDE. Then, an ODE solver is applied in time to close the scheme in time, and in this case, Runge-Kutta (RK) integrators are used. The RK integrator currently being used for the dynamics is a five-stage, third-order accurate integrator from [4] that provides a very large Maximum Stable CFL (MSCFL) value for a large time step. The tracers use a three-stage, second-order accurate Strong Stability Preserving (SSP) RK method [2], and the reason for using a SSP method is that it guarantees that if the spatial operator is monotonic, then they temporal integrator will not introduce monotonicity violations of its own.

2.2 Runtime Characterization

2.2.1 Throughput and Scaling

Climate simulation has a unique and unfortunate constraint not experienced by many scientific fields in that it must be run for very long lengths of simulation time. Many climate runs must complete on the order of 100 years to give useful results and statistics, and multiple scenarios with different anthropogenic forcings must be completed. The current simulations that use an average distance of about 28km between grid points require a dynamics time step on the order of a minute. This means that we must take roughly 50 million time steps before the simulation is completed. The physics are evaluated on a time step of half an hour, meaning roughly two million physics time steps must be taken. In order for the results to be obtained in a reasonable amount of time, the climate community has converged on a widely accepted throughput requirement of about five Simulated Years Per wallclock Day (SYPD).

There are significant problems with having such a constraint. If one holds the throughput as a constant, which we generally must, then each 2× refinement in horizontal grid spacing results in an 8× increase in the amount of work to be performed, 4× of which comes from spatial refinement in two spatial dimensions, and another multiple of 2× that comes from being forced to refine the time step with the grid spacing, since CAM-SE uses an explicit SE method. However, there is only 4× more data / parallelism available because the time dimension is dependent. This means that in order to achieve the same throughput, the refined simulation will have to run on 8× more compute nodes with only 4× more data. Thus, the amount of data and the amount of work per node cuts in half with each 2× spatial refinement. And this is in a perfect world. In reality, since scaling is not perfect, more than 8× nodes will be required in order to keep the same throughput, and we are left with *less than* half the work per node for each 2× refinement.

What this means, practically, is that the CAM-SE code, when run in production mode, is always strong scaled to a significant extent. This means that MPI data exchanges are always consuming a significant amount of the runtime. Coming exascale platforms will interact with this reality in an interesting way. First, it appears that exascale computing will, on the whole, prefer "fatter" nodes, which we take here to mean that aggregate memory bandwidth on a node will grow faster than the number of nodes (a node being a collection of processing elements separated from other nodes by network interconnect). The only benefit this really buys us is that we can spend comparatively more money on interconnect for improved bandwidth. Latency, however, appears to be remaining stagnant unless a transformative technology comes along.

Also of significant concern on current architectures is the regrettable dependence on a high-latency and low-bandwidth PCI-express bus. First, the latency is an order of magnitude higher than the latency of nearest-neighbor MPI communication. Second, the bandwidth of PCI-e is roughly the same as the interconnect used by MPI, meaning the MPI bandwidth costs are doubled. It appears that while future technologies will significantly increase the bandwidth, the latency, again, will remain mostly the same. Thus, smaller transfers are increasingly penalized.

2.3 Code Structure

2.3.1 Data and Loops

CAM-SE uses spectral elements only in the horizontal direction, so a node contains `ntimelevels` time levels of `nelemd` columns of elements with `nlev` vertical levels, each with each of which has `np`×`np` basis functions coefficients. These form the fundamental dimensions of the code and also the loop bounds. For tracer transport, there are also `qsize` tracers that need to be transported, creating an additional dimension for that portion of the code. For most of the code, loops over each of these dimensions provides data-independent work that is relatively easy to vectorize, especially on the GPU.

The data is laid out such that `np`×`np` is the fastest varying dimension, followed by `nlev`, then `qsize` (for tracers only), followed by `timelevels`, followed by `nelemd`. A single column of elements for all time levels is stored in a FORTRAN derived type called `element_t`, which also contains the geometric, basis function, and differentiation data that is used in the SE method for that element. As expected, the looping structure mirrors the data layout with looping over elements as the outermost, followed by tracers, then vertical levels, and finally the basis coefficients.

The practice in the original CAM-SE code is to structure the code largely into two groups of tightly nested loops. The outer tightly nested loops are those over elements, tracers, and vertical levels. The next tightly nested loops are those over the basis functions, and these loops are often inside routines that calculate various derivative-based quantities. This is advantageous in terms of cache locality since often times, the same data is being accessed as the code progresses from one routine to another.

2.3.2 OpenMP

For sake of efficiency, the choice was made at the creation of CAM-SE to structure the OpenMP implementation using parallel regions instead of loop-level OpenMP. The reasoning for this was to ensure that thread pools weren't needlessly destroyed and re-created. So, in the initialization of the code, each thread is assigned a range of element columns: `[nets,nete]`, and that thread only operates on those element columns, and generally this works efficiently, the main exception being the packing and unpacking of element edge data to and from process-wide buffers.

There are difficulties with this when performing a GPU re-factoring of the code, however. At least at the time the code was re-factored using CUDA and then later in OpenACC, poor performance was experienced when multiple threads were launching kernels on the same process and device. Therefore, a thread master region had to be generated for every section of GPU code between MPI boundary exchanges, which themselves cannot be inside master regions.

There is also the option of using all MPI, and simply using the CUDA MPS server (previously called "proxy"), and this works well for many codes. However, since CAM-SE is significantly strong scaled, there ends up being too little work per kernel call for this to be efficient in practice.

2.3.3 Pack, Exchange, and Unpack

Averaging of the element edge data ends up being by far the most expensive operation in the CAM-SE tracers and dynamics because this is where the majority of the data movement occurs, both to and from on-node DRAM and over network interconnect.

Packing essentially just places element edge data into predefined locations in a process-wide buffer. Data is placed into this buffer in such a manner that data moving from one process to another is contiguous in memory so as to minimize the number of MPI data transfers and reduce latency exposure. Next, the boundary exchange takes data from the process-wide buffer that must go to a neighbor and performs an `MPI_Isend` on that data. It then performs an `MPI_Irecv` on data it is receiving and places that data into a receive buffer. Next, the boundary exchange routine copies internal data from the edge buffer to the receive buffer so that all data is in the receive buffer. Then, finally, the unpack routine takes data from predefined locations, sums them up, and replaces its value with that sum. Later a multiplier will be applied to scale this sum down into an average.

2.3.4 Bandwidth and Latency in MPI Communication

For production runs, MPI messages in the dynamics between adjacent elements will generally at most be on the order of 10-20Kb, while messages between corner elements will be at most 1-2Kb. If we assume, for instance, that nearest neighbor interconnect latency is roughly a microsecond, and the bandwidth is roughly 6 GB/sec (which is true for OLCF's Titan), the bandwidth costs for adjacent element messages is only three times larger than the latency costs, and for corner elements, latency clearly dominates. When running larger problem sizes, often times the MPI communication is not nearest neighbor, and latency costs increase. For this reason, the dynamics are extremely sensitive to interconnect latency costs. For the tracer transport routines, however, the message sizes are increased by a factor of about 40 because there are currently 40 tracers being evolved, which can all be sent together. Therefore, the tracer transport is bandwidth bound in the MPI rather than latency bound. This means one must take different optimization strategies when refactoring the dynamics versus the tracer transport.

2.4 Previous CUDA FORTRAN Refactoring Effort

Previously, CUDA FORTRAN was used to refactor the atmospheric tracer transport only to GPUs [6, 1, 5]. This, generally speaking, was a poor design choice, but compiler implementations of the OpenACC standard were nowhere near ready for use in a real code due to bugs and performance problems. There were software maintainability problems with using CUDA in a FORTRAN context:

- The syntax of CUDA FORTRAN is so foreign to the existing FORTRAN standards that the resulting code is usually unrecognizable. The chevron syntax is truly bizarre.
- It requires separating out each kernel into unnecessary subroutines that will not be re-used and are now distant from their previous context in the code.
- It changes every kernel from a simple in-file loop structure to a kernel call to a subroutine.
- At the time, it required breaking all variables used in kernels out of their derived types.
- There are no longer any actual loops, which is quite foreign to any domain science programmer and makes the subroutine much more difficult to read.

- CUDA Shared Memory had to be used in order to gain efficiency in the kernels, and being ideologically a local temporary variable, it is very confusing to see a local variable used in a context that looks global with CUDA FORTRAN's "`__syncthreads()`" used in between.
- There is no incremental refactoring to CUDA FORTRAN once the code is placed into a kernel, which makes debugging more difficult.
- The same code will not run on CPU and GPU like it does in OpenACC, and this also makes debugging significantly more difficult.

The original CUDA FORTRAN refactoring gave good performance, but it turned out that the codebase was still somewhat in flux. This meant that changes needed to be propagated into the CUDA code, which was quite difficult to support. The refactoring effort had to be re-performed once because of structural changes that didn't agree well with the previous CUDA structure. For this reason, the GPU code was transferred into OpenACC code for better software maintainability.

2.5 OpenACC Refactoring

2.5.1 Thread Master Regions

As mentioned before, we made a choice to use a master thread for all GPU kernel activity instead of calling kernels within threaded regions for several reasons. First, the amount of work per kernel call was typically too low to gain efficiency on the GPU when running from threads, and therefore aggregating all threads to a single kernel call was preferred. Also, when running threaded, even with separate asynchronous ids for each CPU thread, we found that the kernels were being serialized on the device and not running concurrently.

This process created no shortage of difficulties in the re-factoring process, as creating master regions in an OpenMP parallel region code can be very prone to producing bugs. As an example, many array temporaries are passed wholly into subroutines, where it is assumed they only have memory for the range `[nets,nete]`. Once inside a master region, it is easy to forget to add this array slice to the subroutine input. Also, the thread's element bounds are often passed into routines, and they must be changed to `[1,nelemd]`. This process begins on a small routine bracketed by barrier statements and master / end master statements inside them. Then, the next small section of code is added into the master region, and so forth until the entire section of code being ported was included in the master region.

2.5.2 Breaking Up Element Loops

For several reasons, the long loop bodies inside the element and vertical level loops were broken up. First, OpenACC was having a lot of difficulty with the use of the routine directive at the time, especially for functions rather than subroutines, and the compiler also was unable to inline the routines. Second, it's important not to let the body of a kernel get too long because eventually, register pressure sets in, and the occupancy of the kernel becomes poor, as does the performance. Therefore, routines that previously only iterated over the basis functions within one element and one vertical level were transformed so that they iterate over all loop indices in the problem.

This, in turn, required several other transformations. First, small array temporaries that previously only dimensioned as `(np,np)` had to be changed into globally sized variables. This increased the memory requirements of the code, but it isn't a problem for ACME, which runs strong scaled to such an extent that most of the on-node memory still remains unused. This is another opportunity for bugs to be introduced because many FORTRAN compilers allow (I believe erroneously) one to use fewer subscripts than actually exist in the declared array. Therefore, failing to add the extra subscripts to something that used to be a small array temporary will produce a bug.

Also, this requires pushing loops down the callstack, which is a common operation when refactoring codes to run on GPUs. Instead of having each of the derivative-type routines operate only on the basis functions of a single element and vertical level, they are now transformed to work on all levels of parallelism up to the element level. This requires adding more dummy parameters to be passed in because they routine needs to know, for instance, what time level to work on or what range of elements to loop over. This, in turn, leads

to another transformation requirement in order to allow the subroutines to be re-usable, and this will be discussed in the next subsection.

2.5.3 Flattening Arrays for Reusable Subroutines

There is an issue that arises when pushing loops down the callstack while using a derived type to hold each element's data. When the subroutine must now loop over elements, it must know the actual name of the variable within the element derived type being operated upon. Previously, one could simply pass, for instance, `elem(ie)%state%T(:,:,k,n0)` to the routine, where `elem` is an array of the element derived type, `k` is the vertical level index, and `n0` is the time level being operated upon. This is because the routine doesn't operate over the element loop. However, once the element loop is pushed down into the routine in question, one must pass all of `elem(:)`, and inside the routine, `elem(ie)%state%T` would have to be explicitly named. At this point, that routine is no longer reusable for other variables besides `T`.

It is poor software engineering practice to create multiple routines that do identical work, and therefore it was desirable to make these routines reusable for any particular data. In order to accomplish this, we created a flat, global array, called `state_T(np,np,nlev,timelevels,nelemd)` for this example. Then, in order to make sure the rest of the code doesn't need to know about this new flattened array, we use a pointer inside the element data structure, and we point into this array so that either syntax may be used for the CPU code at least. Then, `state_T` can be passed into the routine in question, or any other flattened array. We only flattened the arrays that needed to be passed into a reusable subroutine.

This, in turn led to yet another potential for bugs, and this was difficult to find at times. Because the form of `elem(ie)%state%T` is now a pointer, and at least for the PGI compiler, only the flattened array syntax, `state_T(...)` is allowed in any of the OpenACC code, be it a data statement or a kernel. This means one must remember to add the flattened array to the present clause of the kernel along with elem to ensure correctness, and one must replace all kernel references to that array with the flattened expression, including previously ported kernels (which is where the problem usually arose). Probably the worst part of it all was that one does not always get an invalid address error in the kernel when using the derived type syntax inside a kernel. Sometimes it can give the right answer, and then later, it will give an invalid memory address error. The intermittency of this error lead to difficulty in finding bugs in the refactored code.

2.5.4 Loop Collapsing and Reducing Repeated Array Accesses

For the majority of the kernels, some relatively simple optimizations gave good enough performance. First of all, because there are always four to five levels of tightly nested loops in each of the kernels, loop collapsing is absolutely essential. This is mainly because one cannot nest multiple "`!$acc loop gang`" or "`!$acc loop vector`" directives when using the "`parallel loop`" construct in OpenACC. We hope this restriction will be removed in later versions of OpenACC.

Also, the "`kernels`" construct is not relied upon in the ACME code because it generally gave very poor performance. The `kernels` construct is one in which most of the modifiers are simply suggestions to the compiler about what is parallelizable. It leaves a lot of room for the compiler itself to make choices about where to put the parallelism for various loops. For CAM-SE, the compiler usually made very poor decisions. The `parallel loop` construct is much more like the OpenMP parallel do construct, in that it is highly prescriptive rather than descriptive, giving the user a larger amount of control. Therefore the `kernels` construct has been abandoned, and the `parallel loop` construction is relied upon in all of the kernels.

The last general optimization strategy is to replace repeated array accesses with a temporary scalar variable. Presumably, this makes the compiler more likely to place the scalar into register rather than repeatedly accessing the array over and over again from DRAM. It is our opinion that this is a performance bug in the compiler because the compiler should already be placing repeated array accesses into register and keeping it there. But for now, this is an optimization technique that provides a lot of benefit.

2.5.5 Using Shared Memory and Local Memory

As mentioned earlier, because the SE method is essentially 1-D sweeps of matrix-vector products, at least at its core, these kernels stand a lot to gain from the use of CUDA Shared Memory. The inner parts of these routines' loops often look somewhat like the following:

```
do ie = 1 , nelemd
  do k = 1 , nlev
    do j = 1 , np
      do i = 1 , np
        dsdx00=0.0d0
        dsdy00=0.0d0
        do s = 1 , np
          dsdx00 = dsdx00 + deriv(s,i)*glob_1(s,j,k,ie)
          dsdy00 = dsdy00 + deriv(s,j)*glob_1(i,s,k,ie)
        enddo
        glob_2(i,j,1,k,ie) = c1 * dsdx00 + c2 * dsdy00
        glob_2(i,j,2,k,ie) = c3 * dsdx00 + c4 * dsdy00
      enddo
    enddo
  enddo
enddo
```

We have found that the most efficient way to run this on the GPU is to thread the outer four loops on the GPU and to leave the inner loop sequential. What this means is that (1) the kernel is accessing `glob_1` and `deriv` multiple times in each thread and (2) neither `glob_1` nor `deriv` are being accessed contiguously in the GPUs DRAM memory. Since the GPU is actually threading the `i` and `j` indices, the presence of the `s` index in those arrays means they are not being accessed contiguously. In order to alleviate this problem, these arrays are both put into CUDA Shared Memory.

However, in order to do this, the vertical levels loop must be tiled. When run in production mode, `np` is always set to four, meaning the `i` and `j` loops have 16 iterations together. Also, 72 vertical levels are currently being used. For most cases, best kernel performance is obtained by using 128 to 256 threads within an SM. Therefore, the `i` and `j` loops do not have enough parallelism to fill an SM, and the `i`, `j`, and `k` loops together have too much parallelism. Therefore, with kernels that need to use Shared Memory, the vertical level loop is tiled to produce something that looks like:

```
do ie = 1 , nelemd
  do kc = 1 , nlev/kchunk + 1
    do kk = 1 , kchunk
      do j = 1 , np
        do i = 1 , np
          k = (kc-1)*kchunk + kk
          if (k <= nlev) then
            dsdx00=0.0d0
            dsdy00=0.0d0
            do s = 1 , np
              dsdx00 = dsdx00 + deriv(s,i)*glob_1(s,j,k,ie)
              dsdy00 = dsdy00 + deriv(s,j)*glob_1(i,s,k,ie)
            enddo
            glob_2(i,j,1,k,ie) = c1 * dsdx00 + c2 * dsdy00
            glob_2(i,j,2,k,ie) = c3 * dsdx00 + c4 * dsdy00
          endif
        enddo
      enddo
    enddo
  enddo
enddo
```

With this loop, on can now place `glob_1(:,:,(kc-1)*kchunk+1:kc*kchunk,ie)` and `deriv(:,:)` into Shared Memory on the GPU, and the loops over `i`, `j`, and `kk` can be collapsed down and placed within an SM with the "`vector`" loop clause modifier, and the `kc` and `ie` loops can be collapsed down and distributed over SMs with the "`gang`" loop clause modifier. Routines of this form will typically use `kchunk=8` because

it gives a vector length of 128, which often performs best on the K20x and K40 GPUs at least. The `cache` clause is used to place the above variables into CUDA Shared Memory. Also note that on the CPU, this code will not vectorize because of the if-statement inside the inner loop. On the GPU, this is not a problem because if `k` gets larger than `nlev` at any point, the cores will simply no-op. During runtime, you do not even notice the presence of the if-statement on the GPU.

We did encounter a problem when trying to cache contiguous chunks of a global variable like mentioned in the previous paragraph. The PGI compiler has a performance bug wherein the correct amount of Shared Memory is allocated, and it is used correctly, but it is loaded multiple times from DRAM, causing significant performance degradation. This issue is being addressed, but currently, we are having to create our own array temporary of the size `(np,np,kchunk)`, declare it `private` at the `gang` level, and then use the `cache` clause on that variable above the collapsed vector loops. It is a tedious process, and it requires knowing about the array temporary at the `gang` loop levels where the code doesn't actually need to know about the array. Once this performance bug is fixed, the code that uses Shared Memory will certainly be simpler.

2.5.6 Optimizing the Boundary Exchange for Bandwidth

The tracer transport section of CAM-SE has much larger MPI transfers than the dynamics, and it turns out that these MPI transfers experience more runtime cost due to bandwidth than due to latency. Also, given that on OLCF's Titan, the PCI-e bandwidth is roughly the same as that of MPI, it is important to hide as much of that as possible so that the bandwidth costs are not fully tripled because of GPU usage.

- Pre-post the `MPI_Irecv` calls
- Call "`!$acc update host(...)`" of each stage's data in its own asynchronous ID ("`asyncid`")
- Begin a polling loop
  - Once a stage's data is on the CPU (tested by "`acc_async_test`")
    - Call the `MPI_Isend` for that stage
  - Once a stage's `MPI_Irecv` is completed (tested by "`MPI_Test`")
    - Call "`!$acc update device(...)`" for that stage's received data
- Call "`!$acc wait`" to ensure all data is finished being copied

In this way, PCI-e copies from device to host, PCI-e copies from host to device, and MPI transfers are all able to overlap with one another.

2.5.7 Optimizing the Boundary Exchange for Latency

For the dynamics, however, the issue is not bandwidth but latency. The aforementioned strategy would basically end up with a ton of PCI-e traffic, each call incurring a measure of latency. Therefore, the strategy for minimizing latency costs is to simply: (1) pre-post the `MPI_Irecv` calls; (2) call "`!$acc update host(...)`" for each stage asynchronously followed by an "`!$acc wait`" ; (3) call `MPI_Isend` for each stage; and (4) call "`!$acc update device(...)`" for each stage asynchronously followed by an "`!$acc wait`". The extra calls to "`acc_async_test`" cost more in terms of latency than they are worth, since there isn't really any bandwidth to overlap in the first place. Also, calling the PCI-e copies via "`!$acc update`" asynchronously and then waiting after the fact helps avoid noticing the PCI-e latency for each call.

2.5.8 Use of CUDA MPS

It turns out that it is more efficient to use a combination of MPI tasks and OpenMP tasks on a node rather than simply using all OpenMP. We aren't entirely sure of the reason, but it appears to be the case that running multiple MPI tasks on the same GPU gives a natural overlap between PCI-e and MPI costs that isn't being effectively exploited by the technique in the section titled, "Optimizing the Boundary Exchange for Bandwidth." The optimal way to run on OLCF's Titan is to use four OpenMP threads per MPI task (so that non-ported sections of the code like the physics still run on all of the CPU cores) and four MPI tasks per node. Since there are multiple MPI tasks running per GPU, this also requires the use of CUDA MPS.

2.6 Optimizing for Pack, Exchange, and Unpack

In the past, there was an optimization over all of the pack, exchange, and unpack calls. First, during initialization, it would be determined which elements on the node are internal (meaning they have no dependency on MPI) and which elements are external. Then, when pack, exchange, and unpack are called, the external pack is first called asynchronously in one thread, and then the internal pack and unpack are called asynchronously in another thread (since there is no exchange for internal elements). Then, after waiting on the first thread, the boundary exchange is performed as normal. In this way, the internal pack and unpack routines, which are costly, are overlapped with the PCI-e copies and the MPI transfers.

However, in practice, this is of little use for production runs because they are strong scaled to the point where there aren't usually any internal elements in the first place. Therefore, this practice is not currently used. In the future, when more work per node is expected, this technique will likely be reinstituted.

2.7 Testing for Correctness

Testing for correctness is one of the most crucial parts of the refactoring process, and it is also one of the most difficult to perform in an efficient way. As is true with most production science codes, CAM-SE can be run in many different ways with different runtime options. This can certainly make things difficult when refactoring the code, and for this reason, a single configuration was chosen at a time, and any other configuration is caused to abort with an error message until that code can be properly tested during the refactoring effort. Once a given configuration is ported, then more options can be added one at a time while exercising that specific code in a test between development cycles.

It's significantly easier to find and fix a bug when only a small amount of code could have caused it. For this reason, it's crucial to have a fast running test that can be performed frequently. This was an easy situation for the tracer transport routines, as they are linear in nature, thus giving machine precision similar results as the CPU code. For the tracers, a simple recompile, test run, and diff on the output would only require a couple of minutes at most. There was a small amount difficulty getting the code to work for a generic number of vertical levels, but this was not too cumbersome.

The dynamics were another story entirely for three main reasons: (1) there are more options to exercise at runtime than for the tracers; (2) the dynamics are non-linear in nature, which causes small differences to grow exponentially; and (3) a lot of bugs in the dynamics do not show up after a single time step.

Regarding the first issue, there are options chosen in the full CAM-SE code that are not available or not default in the standalone "dynamical core" (dynamics + tracer) code. Also, the CAM-SE standalone code is fairly deprecated, meaning that to exercise CAM-SE as a whole, it really needs to be run inside of a full ACME compilation. Data models are used for the ocean, land ice, and sea ice, but the land model has to be active when the atmosphere is active when running inside ACME. Compiling the land model and CAM-SE takes much more time than simply compiling the standalone dynamical core, which lengthens the time spent testing between development cycles.

Regarding the second issue, climate is inherently a statistical science and not a deterministic one, and this owes itself to the fact that small differences in a fluid state will amplify exponentially in simulation time due to the non-linearity of the PDEs that describe fluid flow. This is precisely why there are limits to our weather forecasts, in fact. What it means for correctness testing is that there isn't really a truly robust way to test for correctness in a quick manner. Therefore, we developed as robust of a smoke test as possible that could be run in a reasonable amount of time. First, the CPU code is run with different compiler flags (-O0 and -O2 in our case) for a few time steps, outputting every time step. Next, the global norm of the absolute difference is computed at each of the time steps. This establishes the expected baseline of answer differences over several time steps that are solely due to machine precision changes, and it only needs to be done once. Finally, since the GPU runs should only cause machine precision differences, the global norm of the absolute difference between GPU and CPU must be similar to that of the difference between -O0 and -O2.

Finally, the reason we do this over several time steps (eight ended up being good enough) is that some bugs do not manifest in this smoke test after only one time step. What this means, in total, is that the smoke testing between development cycles to most easily find and fix bugs during the refactoring process took on the order of 15 minutes. It meant that the porting of the dynamics was a very slow process compared to the tracer transport. However, in the end, there was a large amount of confidence that the solution was indeed correct.

The only truly robust way to test for correctness with machine precision changes without having to revalidate the code against observations is to run a suite of ensembles to establish a baseline, and then to run a suite of ensembles with the new version. Then, statistical tests can be performed to ensure that the two sets of ensembles were, in fact, sampled from the same population, not just for globally and annually averaged quantities but also with spatial and intra-annual variations. This is a capability under current development in ACME.

3 Nested OpenMP for ACME Atmosphere

3.1 Introduction

Accelerated Climate Modeling for Energy (ACME) is a global climate model with coupled components for atmosphere, land, ocean, sea-ice, land-ice and river runoff. Model Coupling Toolkit combines individual components into various component sets that are focused on atmosphere, ocean and sea-ice, land, or on fully coupled global climate issues such watercycle and biogeochemistry. Each compset together with the discretization grid for its components contains a different computational workload. Time integration of the workloads on the 80-year time scale of 40-year hindcast and 40-year forecast can answer various climate simulation questions. Computational performance of a workload is measured with a throughput rate of model simulation years per wall-clock day (SYPD). Our goal is the optimization of the throughput rate of these workloads on peta- and pre-exascale computing machines.

Currently, the highest discretization grid in ACME provides a resolution of 25 km between gridcell centers. At this resolution, the primary throughput rate limiting component is atmosphere and in particular the time-integration of atmospheric dynamics by the High-Order Methods Modeling Environment (HOMME) dynamical core of ACME. Improving scalability and performance of HOMME can lead to the overall throughput acceleration of ACME's watercycle and biogeochemistry climate simulations.

In this work, we describe improvements to OpenMP-based threading and vectorization in HOMME. We use nested threading to increase the current scaling limit of one gridcell (a HOMME element) per core. Nested threads iterate over sub-elemental data structures and lead to performance improvements. In addition, we discuss OpenMP-based vectorization of iterations over innermost data structures. Together with other code refactoring to help compilers generate efficient code, vectorization leads to more efficient use of on-chip hardware resources and better performance. Overall, combination of the OpenMP-based parallelization techniques exhibits up to 40% overall improvement in the throughput rate of high-resolution atmosphere compset on ALCF's Mira.

The outline of our discussion is as follows. In Section 3.2, we overview the existing algorithmic structure of HOMME's dynamics and summarize the hybrid MPI+OMP parallelization approach. In section 3.3, we present the programming approach of nested threading. Section 3.4 discusses software engineering issues affecting performance engineering. In Section 3.5, we conclude with performance benchmarking results.

3.2 Algorithmic Structure

The discretization grid in HOMME is a cubed sphere with six cube faces and ne number of elements per cube edge for a total of $6ne^2$ elements. The high-resolution 25km grid has `ne`=120 elements on an edge and a total of 86400 elements.

Each element is further decomposed into vertical levels to represent the vertical dimension. Finally, each 3D element slice is divided into two-dimensional sub-elemental columns. The hierarchical decomposition of an element's state lends itself naturally to the Structure-Of-Arrays data storage. Thus the layout of element data along with the typical parameters used in ACME have the following form in Fortran:

```
1 !Vertical Levels
2 integer, parameter, public :: nlev = 72
3 !Number of Tracers
4 integer, parameter, public :: qsize = 35
5 !Number of basis functions in 1-D per element
6 integer, parameter, public :: np = 4
7 !Number of time levels
8 integer, parameter, public :: timelevels = 3
```

```
type, public :: elem_state_t
  !Temperature
  real (kind=real_kind) :: T(np,np,nlev,timelevels)
  !Tracer Concentration
  real (kind=real_kind) :: Q(np,np,nlev,qsize)
  ...
end type elem_state_t
type, public :: element_t
   type (elem_state_t) :: state
   ...
end type element_t
```

For parallel efficiency, the mapping between decomposed data and parallel MPI processes is done at the level of HOMME elements. This is done to reduce the inter-process communication, because only elements that share an edge or a corner need to exchange boundary data. Most of the computation is done within an element and takes advantage of low intra-process shared memory latencies.

For flexibility, elements can also be mapped onto OpenMP threads making MPI ranks and OMP threads interchangeable in element processing. During initialization, each rank and thread is assigned the starting and ending global indices of owned elements.

HOMME provides a caller (e.g. CAM component) procedures to initialize its state and then drive the integration forward in time with sub-cycling to account for tracer advection, vertical remapping and fluid flow sub-steps within a coarse dynamics step, which is interleaved with physics timestepping. All subroutines operating on an element's state are driven by the `dyn_run` procedure abstracted below with relevant aspects.

```
subroutine dyn_run( dyn_state, rc )
#ifdef HORIZ_OPENMP
  !$OMP PARALLEL NUM_THREADS(nthreads), DEFAULT(SHARED), &
  !$OMP& PRIVATE(ithr,nets,nete,hybrid,n)
#endif
#ifdef COLUMN_OPENMP
  call omp_set_num_threads(vthreads)
#endif
  ithr=omp_get_thread_num()
  nets=dom_mt(ithr)%start
  nete=dom_mt(ithr)%end
  hybrid = hybrid_create(par,ithr,nthreads)
  do n=1,nsplit ! nsplit vertical remapping sub steps
    ! forward_in_time Runge_Kutta, with subcycling
    call prim_run_subcycle(dyn_state%elem, dyn_state%fvm, hybrid, &
                           nets, nete, tstep, TimeLevel, hvcoord, n)
  end do
#ifdef HORIZ_OPENMP
  !$OMP END PARALLEL
#endif
end subroutine dyn_run
subroutine prim_run_subcycle(elem, nets, nete, nsubstep)
  do r=1,rsplit ! rsplit tracer advection sub steps
    call prim_step(elem, fvm, hybrid, nets, nete, dt, tl, hvcoord, &
                   cdiags, r)
  enddo
  call vertical_remap(...)
end subroutine prim_run_subcycle
subroutine prim_step(elem, fvm, hybrid, nets, nete, dt, tl, &
                     hvcoord, compute_diagnostics, rstep)
  do n=1,qsplit ! qsplit fluid dynamics steps
    call prim_advance_exp(elem, deriv(hybrid%ithr), hvcoord,   &
                hybrid, dt, tl, nets, nete, compute_diagnostics)
  enddo
  call euler_step(... step1 ...)
  call euler_step(... step2 ...)
  call euler_step(... step3 ...)
end subroutine prim_step
```

The essential feature of this algorithmic structure is the OpenMP parallel region at the beginning of the dynamics time stepping, which can be disabled with CPP preprocessing for pure-MPI simulation runs. Parallelization over elements is programmed explicitly with first and last element indices corresponding to the

thread number's index range. Inside the subroutines each element-level loop is ranged as `do ie=nets,nete ... enddo` ensuring that the workload of iterating over elements is shared by `nthreads` number of OpenMP threads.

Within the `euler_step` subroutine there are calls to pack (and then unpack) an element's boundary data and exchange it with neighboring elements using a sequence of `MPI_Isend`, `MPI_Irecv` and `MPI_Waitall` calls. If multiple elements are owned by an MPI rank, then only the external boundary data is exchanged and intra-process internal element boundaries are exchanged in shared memory. To reduce the overall number of messages and avoid potential network congestion, only the master thread communicates boundaries owned by all other threads within an MPI rank.

3.3 Programming Approach

Having described the hybrid MPI+OMP parallelization over elements, we now turn to performance improvements in computations over sub-elemental data structures. Since most of the computation time is spent in loops, we focus on two forms of parallelization of loops: nested threading over sub-elemental loops and vectorization of innermost loops.

As a coarse summary of loops in HOMME, we note that there are approximately 3390 loops in ≈56KLOCs (including comments) of 63 Fortran90 source files in HOMME's `src/share` directory. Performance profiling results indicated a relatively flat profile with no hot spots of computation. Instead, there were several "warm" loops that could be targeted by nested OpenMP parallel regions. We used GPTL timers [9] to measure the computation time taken by a loop and added parellel regions to compute-intensive loops. The following listing shows an example extracted from euler_step subroutine.

```
1  real(kind=real_kind) :: Qtens_biharmonic(np,np,nlev,qsize,nets:nete)
2  do ie = nets, nete
3    !$omp simd
4    do k = 1, nlev
5      dp(:,:,k) = elem(ie)%derived%dp(:,:,k) - rhs_multiplier*dt*  &
6                  elem(ie)%derived%divdp_proj(:,:,k)
7    enddo
8  call t_startf('eul_nested')
9  #if (defined COLUMN_OPENMP)
10 !$omp parallel do private(q,k) collapse(2)
11 #endif
12   do q = 1, qsize
13     do k=1, nlev
14       Qtens_biharmonic(:,:,k,q,ie) = &
15               elem(ie)%state%Qdp(:,:,k,q,n0_qdp)/dp(:,:,k)
16       qmin(k,q,ie)=minval(Qtens_biharmonic(:,:,k,q,ie))
17       qmax(k,q,ie)=maxval(Qtens_biharmonic(:,:,k,q,ie))
18     enddo
19   enddo
20   call t_stopf('eul_nested')
21 enddo
```

This example demonstrates three parallelization levels: (1) at the horizontal element level with threads iterating over `ie` elements, (2) at the vertical column level with nested threads iterating on the collapsed loop over tracer `q` and level `k` indices, and (3) at the vertical and sub-column levels with data-parallel SIMD vectorization. Nested threading directive is protected with a CPP macro to disable it for specific system and compiler combinations: e.g. nested threading is not as beneficial on Titan when compiled with PGI compiler compared to OpenACC-based approach. To ensure that nested threads have a sufficiently large workload to justify the overhead of a nested parallel region, the two `q` and `k` loops are collapsed into one combined loop using OpenMP `collapse(N)` clause. Finally, if a compiler does not auto-vectorize innermost loops over `np`×`np` sub-columns that are denoted by Fortran array notation :, we provide an `!$omp simd` hint to the compiler that the annotated loop should be vectorized.

3.4 Software Practices

ACME uses Git versioning system to track software development in a shared repository hosted at `http://github.com`. Developers track their work in git branches that are submitted to component (ATM, LND,

ICE, OCN) integrators in pull requests (PR). Component integrators review PRs and merge them into an integration branch called `next`. Nightly integration tests run on major system and compiler combinations (e.g. Edison+Intel, Mira+IBM, Titan+PGI) to ensure that the new branch does not break existing and tested features. After all tests pass, the new branch is merged into the `master` branch.

Besides the expected checks for absence of build- and run-time errors, ACME uses baseline comparison testing. Simulation tests produce NetCDF output files that capture the state of a simulation at the end of a run. A baseline output is first analyzed for correctness: e.g. results of a hindcast run match observational data. Then, every subsequent run of the same simulation test must produce outputs that are bit-for-bit identical with the established baseline – BFB. If the results of the test on a developer's branch are not BFB, then the branch needs to be re-worked to become BFB. If the developer provides evidence that the new results should be accepted as a new baseline, then the nightly integration testing framework's set of baselines is updated with the new results.

Initial work on nested threading required substantial efforts into ensuring that outputs after adding nested parallel regions were bit-for-bit identical to outputs produced with pure-MPI simulation runs. The most common source of errors that were producing non-BFB results was missed identification of variables that should be `private` within a nested parallel region. Compilers and OpenMP implementations do not auto-detect that a variable is overwritten by multiple threads and such hazards had to be manually inspected and removed.

3.5 Benchmarking Results

To measure the benefits of nested threading we have benchmarked atmosphere-focused component set on Mira. Figure 1 shows results of hybrid MPI+OpenMP benchmarking at various rank and thread combinations including nested threads. We strong-scale the workload of the high-resolution F-compset, which contains $6 * 120^2 = 86400$ elements, from 1350 to 5400 Mira nodes. This corresponds to allocating 64, 32 and 16 element workload on a 16-core 4-way multi-threaded node. Within each workload, we also benchmark various combinations of MPI ranks and OMP threads to determine the optimal throughput rate of simulation years per day (higher is better). Based on the results, we can observe that MPI is more efficient than horizontal OMP threads: i.e. 4×16 clearly outperforms all 1×64 configurations with 1 MPI rank and 64 OMP threads.

In addition to MPI-vs-OMP efficiency comparison, we scale the workload to 8192 nodes. In this mode, there are fewer elements than there are available cores (and hardware threads) and we can use them for nested threading. For example, the $5400 \times 4 \times 4(\times 4)$ configuration uses 4 nested threads to iterate over vertical column and tracer loops and this configuration achieves $\approx 25\%$ speedup over the traditional hybrid MPI+OMP approach. Further, job scheduling on Mira enforces power-of-two partition sizes, which implies that a 5400-node job occupies an 8192-node partition. To fully occupy the partition, we use 5 nested threads in the $8192 \times 4 \times 3 \times 5$ configuration. This achieves the highest throughput rate of 1.22 SYPD, which is a speedup of $\approx 40\%$.

4 Portability Considerations

Portability is one of the most important considerations for our efforts in ensuring ACME continues to efficiently utilize LCF architectures both now and in the future. Portability is something that must be balanced with performance, as one cannot obtain ideal performance on multiple architectures with the exact same codebase. However, it may be that one can eventually obtain suitably good performance with the same codebase.

The real goal is probably better labeled "maintainability" than strict portability. For now, one must use a separate codebase for the CPU and GPU. More than likely Intel's MICs will be roughly similar enough to a CPU that one could unify the CPU and MIC code base with a likely a large amount of refactoring beforehand. However, there are several issues precluding a unified CPU / GPU codebase a present. We hope some of these issues are abated in the future by standards changes to OpenACC / OpenMP as well as improved compiler implementation and hardware features.

For the atmospheric code, we have implemented a CMake target-based handling of the multiple architectures. We have a target for the CPU, and then we have that same target name with "_acc" appended

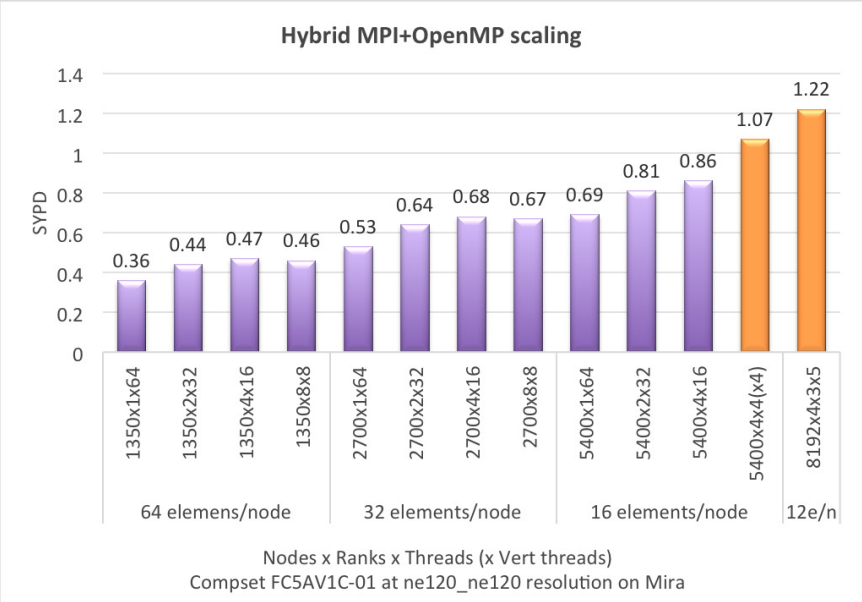

Figure 1: Hybrid MPI+OMP scaling and nested threading on Mira. A workload of 86400 elements is scaled from 1350 nodes to 8192 nodes at various $rank \times thread$ configurations. Results indicate that MPI is more efficient than horizontal OMP. Also, nested threading in the rightmost 2 configurations provides an $\approx 40\%$ speedup at the highest scales.

for the OpenACC version of the code. Both codes look similar enough that it's easy to move changes from CPU development OpenACC code when necessary. All common code shared between the CPU and GPU versions are kept in a shared directory. Then, when something needs to be specialized for the GPU, the base module is used, and only those routines that need altering are re-implemented.

Ideally, this functionality would be implemented using Fortran 2003 classes. However, we and others have found that compiler implementation of recent Fortran standards is quite poor when used inside OpenACC regions (and often even when implemented only for the CPU). We wish to avoid running into those current limitations. Improved compiler implementation coupled with the coming Unified Memory between CPU and GPU (which will greatly improve pointer functionality) should enable the use of modern Fortran inside OpenACC portions of the code.

4.1 Breaking Up Element Loops

One of the main reasons that CPU and GPU code cannot currently share a single source code is that as the amount of code in a GPU kernel gets larger, the register pressure increases substantially. This means that the kernel will have poor occupancy on the GPU and will not execute efficiently. The only remedy for this at present is to break the kernel up into multiple kernels. Using function calls will also ameliorate this, but it doesn't solve the problem, as successive function calls also increase register pressure, albeit more slowly. Also function calls on the GPU are quite expensive because the register page and cache are flushed.

On the CPU one would want to keep an element loop fused together for caching reasons, since the CPU is heavily utilizes keeping cached data resident for avoiding the high latency and low bandwidth of DRAM accesses. The GPU, however, is mainly a vector machine and relies mostly on quick thread switching to hide costs of DRAM accesses, so this is less of a concern on the GPU. Thus, one would not want the broken-up loops to run on the CPU, since it would be less efficient. In our GPU port, we have to branch at the element loop level into separate GPU code in order to break up the element loop for nearly every function call to avoid register pressure problems.

We believe that improvements to compiler implementation of OpenACC compilers that improve register handling would remove this block to a unified codebase.

4.2 Collapsing and Pushing If-Statements Down the Callstack

In the atmospheric code, we have up to five levels of looping: element (ie), tracer index (q), vertical level (k), y-basis function (j), and x-basis function (i). The problem with having so many levels of looping is that with the parallel loop construct in OpenACC, one cannot nest multiple "acc loop gang" directives or multiple "acc loop vector" directives. One can do this with "acc kernel" rather than "acc parallel loop," but we will give reasons in the next subsection as to why this is not a good idea for portable or performant code.

When there are more than three levels of looping, then one must use the OpenACC collapse clause in order to access all of them for threading. In our case, we tightly nest the loops into two groups: a collapsed vector loop, and a collapsed gang loop. One of the issues that comes from collapsing loops is that if statements that were in that loop nesting but not on the inner-most level will have to be pushed down to the innermost level, since collapsing requires tightly nested looping. We have a number of instances where this happens.

There is no penalty on the GPU for doing this, as long as there is no branching within a half-warp. However, on the CPU, the code will not land on the vector units if there are if-statements inside the innermost loop that is being vectorized. Thus, this is another reason why CPU and GPU code must be separate currently.

We believe that allowing nested "acc loop gang" and "acc loop vector" directives would remove this block to a unified codebase.

4.3 Manual Loop Fissioning and Pushing Looping Down the Callstack

It doesn't always look that great to manually fission (or block or tile) a loop. However, we believe that for some codes, it is a wise decision whether or not one is running on an accelerator. The reason is twofold: (1) caching is an important consideration on all architectures, but they are quite different in size between CPU, GPU, and MIC; and (2) effective vector lengths will differ substantially between different architectures. In order to provide flexibility to respect varying cache sizes and vector lengths, it is often helpful to manually fission a loop in the code base.

If there is no real callstack, then this isn't necessary, but rarely is that the case for a scientific code. When there is a callstack present, then it is important to be able to flexibly push and pull looping through the callstack. This is most easily done if the developer chooses a loop (possibly two) to manually fission. Then the routines can loop over any range of that fissioned loop, and the amount of looping assigned to the callee as opposed to the caller can be determined at either compile time or run time.

In our case, the atmospheric code originally was coded for the CPU code, and thus, most of the routines only operated on a single vertical level for a single element and for a single variable, looping only over the horizontal basis functions, which in production runs translates to 16 loop iterations. That isn't nearly enough for the OpenACC vector loop on the GPU. The next loop is the vertical level loop, and there are 72 vertical levels. We cannot push the entire vertical level loop into those routines because this would violate CPU cache and GPU vector loop length requirements. Thus, we chose to manually fission the vertical loop, and this gave two improvements.

First, it allows the same routines to keep the CPU code performing as it currently does while also allowing a much larger amount of looping inside the routine to be used for the GPU. We typically push a block of 8 loop indices of the vertical level into the routines for the GPU, which provides 128 loop indices for the OpenACC vector loop, which is typically ideal or close to it. It would also allow some medium between these two to be specifiec for the MIC. Second, it allows us to size the vector loop correctly for the right amount of data to be cached in L1 cache using Shared Memory. Again, a block of 8 from the vertical loop seems about right for most kernels. Everything outside the vector loop is collapsed down into a gang loop.

4.4 Kernels Versus Parallel Loop

There are to main ways of porting a section of code to an accelerator using OpenACC: the "acc kernels" construct and the "acc parallel loop" construct. The advantage of using kernels is that it theoretically offloads many of the decisions to the compiler, since it is more descriptive in nature than prescriptive. However, we have two problems with the kernels construct. First, we've found that for our code, the compiler rarely makes good decisions in terms of vector and gang loop placement. In fact, it often made such poor decisions that the code would perform over 100× worse than the parallel loop construct. Also, with the compiler

making more decisions, our performance would be more exposed to the compiler version, making it less robust overall.

Second, we eventually are going to transition to using OpenMP 4.x when it becomes available and is relatively mature in compiler implementation. OpenMP has more similarities with parallel loop than with kernels, since it is also prescriptive in nature. Thus, if we eventually wish to transition to OpenMP, it is a better route to go ahead and use the parallel loop construct.

5 Ongoing Codebase Changes and Future Directions

Aside from the atmospheric dynamics and tracers, efforts are underway to port other parts of the code, including the MPAS-Ocean model, a so-called "super-parametrization" for the atmospheric physics, and other parts of the atmospheric physics. These combined would account for the vast majority of the runtime cost of ACME, and they are each planned to be ported using OpenACC and migrated later to OpenMP 4.x.

References

[1] Ilene Carpenter, Rick Archibald, Katherine J Evans, Jeff Larkin, Paulius Micikevicius, Matt Norman, Jim Rosinski, Jim Schwarzmeier, and Mark A Taylor. Progress towards accelerating homme on hybrid multi-core systems. *International Journal of High Performance Computing Applications*, page 1094342012462751, 2012.

[2] Sigal Gottlieb. On high order strong stability preserving runge–kutta and multi step time discretizations. *Journal of Scientific Computing*, 25(1):105–128, 2005.

[3] Akira Kageyama and Tetsuya Sato. âyin-yang gridâ: An overset grid in spherical geometry. *Geochemistry, Geophysics, Geosystems*, 5(9), 2004.

[4] Ingemar PE Kinnmark and William G Gray. One step integration methods of third-fourth order accuracy with large hyperbolic stability limits. *Mathematics and computers in simulation*, 26(3):181–188, 1984.

[5] Matt Norman, L Larkin, Rick Archibald, I Carpenter, V Anantharaj, and Paulius Micikevicius. Porting the community atmosphere model spectral element code to utilize gpu accelerators. *Cray User Group*, 2012.

[6] Matthew Norman, Jeffrey Larkin, Aaron Vose, and Katherine Evans. A case study of cuda fortran and openacc for an atmospheric climate kernel. *Journal of Computational Science*, 9:1–6, 2015.

[7] Todd Ringler, Mark Petersen, Robert L Higdon, Doug Jacobsen, Philip W Jones, and Mathew Maltrud. A multi-resolution approach to global ocean modeling. *Ocean Modelling*, 69:211–232, 2013.

[8] C Ronchi, R Iacono, and Pier S Paolucci. The âcubed sphereâ: a new method for the solution of partial differential equations in spherical geometry. *Journal of Computational Physics*, 124(1):93–114, 1996.

[9] James M. Rosinski. Gptl – general purpose timing library. `http://jmrosinski.github.io/GPTL/`, 2016.

[10] MA Taylor, J Edwards, and A St Cyr. Petascale atmospheric models for the community climate system model: New developments and evaluation of scalable dynamical cores. In *Journal of Physics: Conference Series*, volume 125, page 012023. IOP Publishing, 2008.

[11] MA Taylor, J Edwards, S Thomas, and R Nair. A mass and energy conserving spectral element atmospheric dynamical core on the cubed-sphere grid. In *Journal of Physics: Conference Series*, volume 78, page 012074. IOP Publishing, 2007.

[12] Steven T Zalesak. Fully multidimensional flux-corrected transport algorithms for fluids. *Journal of computational physics*, 31(3):335–362, 1979.

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract number DE-AC02-06CH11357.